D5.4 – Final Report on Proof-of-Concept Activities
Version 1.0

Document Information

Contract Number	676553
Project Website	www.pop-coe.eu
Contractual Deadline	Month 30, March 2018
Dissemination Level	Public
Nature	Report
Authors	José Gracia (HLRS)
Contributors	Sally Bridgwater (NAG), Jonathan Boyle (NAG), Nick Dingle (NAG), Jesus Labarta (BSC), Mathias Nachtmann (HLRS), Wadud Miah (NAG), Amer Shah (AACHEN)
Reviewers	Christian Terboven (AACHEN)
Keywords	Proof-of-Concept

Notices: *The research leading to these results has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No n°676553.*

©2016 POP Consortium Partners. All rights reserved.

Contents

Executive Summary **4**

1 High-level Overview of Proof-of-Concept Activities **5**

2 Summary of Ongoing PoC Activities **6**

3 Report on Activity BAND **8**
3.1 Description of the Application . . . 8
3.2 Previous Assessments and Recommendations . . . 8
3.3 PoC activities . . . 9
3.4 Results . . . 9
3.5 Conclusion . . . 11

4 Report on Activity uhisolver **12**
4.1 Description of the Application . . . 12
4.2 Previous Assessments and Recommendations . . . 12
4.3 PoC Activities . . . 12
4.4 Results and Conclusions . . . 12

5 Report on Activity zCFD **14**
5.1 Description of the Application . . . 14
5.2 Previous Assessments and Recommendations . . . 14
5.3 PoC Activities . . . 14
5.4 Results . . . 15
5.5 Conclusion . . . 16

6 Report on Activity DNS JET **17**
6.1 Description of the Application . . . 17
6.2 Previous Assessments and Recommendations . . . 17
6.3 PoC activities . . . 17
6.4 Results . . . 18
6.5 Conclusion . . . 18

7 Report on Activity Vampire **20**
7.1 Description of the Application . . . 20
7.2 Previous Assessments and Recommendations . . . 20
7.3 PoC activities . . . 20
7.4 Results and Conclusion . . . 20

8 Report on Activity Reveal **22**
8.1 Description of the Application . . . 22
8.2 Previous Assessment and Recommendations . . . 22
8.3 Implementation and Results . . . 22
8.4 Conclusion . . . 23

9 Report on Activity SHEMAT Suite **25**
9.1 Description of the Application 25
9.2 Previous Assessments and Recommendations 25
9.3 PoC activities 25
9.4 Results 26
9.5 Conclusion 27

10 Report on Activity Relational Discovery **28**
10.1 Description of the Application 28
10.2 Previous Assessments and Recommendations 28
10.3 PoC activities and Results 28
10.4 Conclusion 30

11 Conclusions and Lessons Learned **31**

Acronyms and Abbreviations **33**

Executive Summary

This document is the final report on the central service activity of Work Package 5, namely the Proof-of-Concept activities. The aim of the Proof-of-Concept activity is to assist users in implementing complex code refactoring or applying advanced parallelisation techniques. Proof-of-Concept activities may be initiated as a result of performance assessments in Work Package 4, but unlike them, are expected to take significantly longer to conclude.

In the course of the last six months of the project, POP has concluded 9 Proof-of-Concept activities. As reported in the rest of the document, these have resulted in substantial performance increase of more than 20% in 7 cases; thereof in 2 cases even very significant performance increase in the order of 200%. Only 1 activities has resulted in marginal or no performance increase. The proposed changes to the code have already been incorporated in some cases into the production code by the customer; in a few cases, this is still being evaluated.

Thus, over the whole duration of the project, POP has completed 19 Proof-of-Concept activities.

Another 3 Proof-of-Concept activities are still in progress at the formal end of the project. POP partners are committed to complete than as soon as possible. Their current state is briefly summarised in this document.

Some conclusions and lessons learned from PoC activities are presented at the end of the document. In essence, we re-iterate that parallel runtime systems should assume responsibility for performance to a larger degree than they do today. To achieve this, however, they need support from programming models which currently do not allow to express programmers intention in such a way that runtimes can take smart informed decisions.

1 High-level Overview of Proof-of-Concept Activities

The project POP offers three distinct service activities. Two of them, Audit and Performance Plan, are performance analysis services hosted in work package 4. The Audit is an initial assessment of an applications performance characteristics and identification of possible bottlenecks, while the Performance Plan does more detailed root cause analysis of bottlenecks. One of the outputs of theses assessments, in particular from the Performance Plan, is concrete recommendations for changes of the applications code, algorithm or data structures.

In some cases, the recommendations can be implemented with simple code refactoring by a person with average parallel programming skills. In general, however, implementing the recommendation will require complex code refactoring and expert knowledge of parallel programming. Such specialised skills are not available in most groups that do parallel programming. Therefore, POP offers a third level of service called Proof-of-Concept (PoC). During this activity, POP staff will exemplify some of the recommended code refactoring, either directly on the customers application or on a simplified skeleton thereof. The aim is to illustrate the necessary changes and to train the customer in the usage of parallelisation and optimisation techniques.

At project start, we anticipated to do PoC activities mainly on kernels which had been extracted from application, or on mini-apps and mock-ups, rather than working on the customer's actual code. This practise would require less effort of POP staff. However, from the start customers preferred PoC activities to be conducted directly on their code: first extracting kernels/mock-ups and then reincorporating the optimisations into their code-base was rightly seen as large additional effort. Due to the low number of PoC requests, the POP project could afford to invest more effort per PoC activity (see D1.3); thus during the course of the second year we changed procedure to add benefit to the customers of the service and now work on actual customer code whenever possible.

PoC activities are expected to yield best-practise guidelines and input for training material which will be taken up in Work Packages 3 *Community Development*, and 6 *Training and Documentation*, respectively. We also expect some feedback on and suggestions for improvement of practise of performance assessment in Work Package 4 *Analysis*.

The remainder of this document reports on the PoC activities conducted during the last six months of the project. Activities, which have already been concluded are reported individually in Secs 3-10. Three activities are still in progress at the formal end of the project, but will be completed as soon as possible. Their current state is summarised in Sec 2. We make some final observations in Sec 11.

2 Summary of Ongoing PoC Activities

In the last six months of the project, POP Project has concluded nine PoC activities, which are describe in subsequent sections of this document. In addition there are three further PoC activities under way, specifically on the codes *Kratos*, *MNCP*, and a second PoC on *EPW*. Their current state is briefly reported in the following. Finally, one PoC activity was cancelled due to unresponsiveness of the customer.

Kratos The Kratos Performance Assessment identified several issues in the OpenMP version of the Kratos code: load imbalances caused by both NUMA architecture; different locality behaviour for the same code in different cores; and issues with atomic reductions with indirections on large arrays. To address then we suggested different refactorings: parallelising data initialisation to leverage the first touch mechanism; look at renumbering mechanisms to uniformize (and improve) locality across threads; and approaches to eliminate atomics by properly scheduling (using commutative and multidependece clauses in OmpSs) a taskified version of the code.

In the PoC we are advising the customer through the implementation of some of these proposals. The work till now has focused on the data initialisation to address the NUMA architecture, using dynamic scheduling (Guided) as a first approach to address locality caused imbalances and partial approaches to address the atomics issue. Addressing the NUMA issues required some changes in the code caused by limitations imposed by the C++ std::vector data types the code was used. The issue is that allocation of this data type happens to sequentially touch the data thus inhibiting the possibility of leveraging the first touch mechanisms.

We also worked a bit on renumbering schemes although the observed gains from this is only about 10%. We need to compare in detailed the impact on IPC of the new numbering compared to the previous one.

The current global result is that an aggregated acceleration above 2x has been obtained till now. We have to do a detailed analysis of the current behaviour and implement more elaborated mechanism to eliminate the atomics overhead.

MNCP An important load imbalance was the main issue detected in the performance assessment of the application. A proof of concept to address it was suggested. Given the granularity at which this imbalance occurred, an over-subscription approach at the MPI level which in many cases would not be recommended could in this case be a possible alternative. The other alternative would be based on using the MPI+OpenMP version of the code and use the DLB library to reassign cores between processes and dynamically balance the load. The customer had only been using the MPI version of the code they had built on their machine from the sources provided by the application developer. As a first step for the hybrid approach they tried a pure OpenMP version distributed in source code and observed a 3x speed up over the MPI version in a single node. There were still some differences in performance between the pure OpenMP version compiled by the customer and the binary distribution provided by the developer the should be investigated.

The MPI+OpenMP source code happened to be different and it came out not to compile properly with the infrastructure available at the customer site. This is acknowledged by the developer. The proper compilation infrastructure is available at BSC but given that the source code requires a special license from the developer, we started the process to apply for such license, but the process seems to be very slow and still we have no answer. We expect to be able to perform the evaluation when this process settles.

EPW The EPW (Electron-Photon-Wannier) code within Quantum ESPRESSO was already assessed in POP studies, and a performance problem when writing the final simulation state rectified in an initial PoC. A further PoC was proposed to investigate whether a new bottleneck when repeatedly reading a large file in parallel could be ameliorated. Since the previous allocation of computation time on ARCHER was exhausted, the PoC is starting with assessment and investigation of the (new) problem on MareNostrum4 (which is now used for development). Work will continue when POP access to MareNostrum4 is restored.

3 Report on Activity BAND

Keywords: MPI; parallel complex matrix-matrix multiplication

3.1 Description of the Application

BAND is a density functional theory (DFT) code which uses atomic orbitals in periodic DFT calculations. BAND is written by SCM (Software for Chemistry & Materials), an Amsterdam-based computational chemistry software company, and is part of their ADF Modelling Suite.

3.2 Previous Assessments and Recommendations

BAND had previously been studied during a POP Audit and two POP Performance Plan studies, one of which analysed performance of complex matrix-matrix multiplications, which were subsequently improved in this Proof of Concept work. Each complex matrix is stored in two POSIX shared memory arrays of reals, which are replicated on each compute node, with two methods available for the computation:

1. In the shared memory array multiplication method, each process computes a block of the result matrix, via four calls to the BLAS dgemm subroutine.
2. In the distributed memory array multiplication method, the shared memory real arrays are converted to distributed complex arrays, and the PBLAS pzgemm subroutine computes the result.

Both methods then use MPI communication to convert the result data to shared array storage.

For the shared array multiplication method, the main issues identified were poor instruction and IPC scalability within dgemm, and poor transfer efficiency. The recommendations were:

- Remove unnecessary MPI_Allgatherv calls by reordering the algorithm.
- Overlap MPI data transfer with computation.
- Test the following, and implement if beneficial:
  - Different partitionings of the multiplication using dgemm.
  - Multithreaded dgemm or zgemm in place of serial dgemm.

For the distributed array multiplication, the issues were poor transfer efficiency and load balance. The recommendations were to test the following, and implement if beneficial:

- Hybrid (OpenMP+MPI) pzgemm in place of MPI pzgemm.
- Improved load balance of pzgemm.

3.3 PoC activities

The first phase of the PoC work used stand-alone code to identify the optimal strategy for use of BLAS and PBLAS in the computation. This work identified:

For the shared memory array multiplication:

- Significant improvements to serial dgemm performance if the output matrix is close to square, rather than rectangular.
- No significant benefit from multithreaded dgemm or zgemm in place of dgemm.

For distributed array multiplication:

- No benefit using hybrid (MPI + OpenMP) pzgemm in place of MPI pzgemm.
- No benefit from improved load balance of pzgemm.

As there were no identified improvements to the distributed array multiplication, the second phase of work concerned only the shared array multiplication, and implemented the following within the BAND source code:

- Each process computes the real and imaginary components of the result matrix before communicating this data via MPI_Allgatherv. This reduces the number of MPI_Allgatherv calls from four to two, and halves the corresponding data transfer.
- The dgemm computation is modified so columns of one input matrix are partitioned over compute nodes, and rows of the second input matrix are partitioned over the processes on each node. This improves the squareness of the output matrix, and thereby improves the computational scaling.
- The first MPI_Allgatherv call is overlapped with dgemm, to improve transfer efficiency.
- The POSIX shared arrays used for temporary storage are allocated once and reused.

The last modification listed above had not been anticipated during the earlier work, but was motivated after noticing calls to MPI_Allgatherv and dgemm using freshly created shared arrays are typically twice as slow, compared to using of arrays which had previously been used.

3.4 Results

The original and new code were compared on the Marconi Broadwell system, which have 36 core compute nodes. The new version of the shared array multiplication showed significant improvement. This can be seen when comparing Extrae timelines (Figure 1) for the original and the optimised code below, where both timelines use the same time scale. In the optimised code there is significantly less time within useful computation (blue Running in the figure) and also within MPI_Allgatherv (orange Group Communication on the first process of each node). The overlap of group communication and useful computation is also obvious.

The plot in Figure 2 shows the scaling for a single call to the original and improved subroutines. On eight compute nodes the speedup for the new code is around four times that of the original. The new code scales reasonably well on 8 compute nodes, whereas the original code showed no significant speedup beyond a single node.

The POP metrics verified the improvement in the new code. For example, for matrices of size 3200x3200 on eight compute nodes, the computational scaling increased from 0.41 to 0.79 due to improved IPC scaling, and the parallel efficiency increased from 0.34 to 0.57, due to improved serialisation and transfer efficiencies.

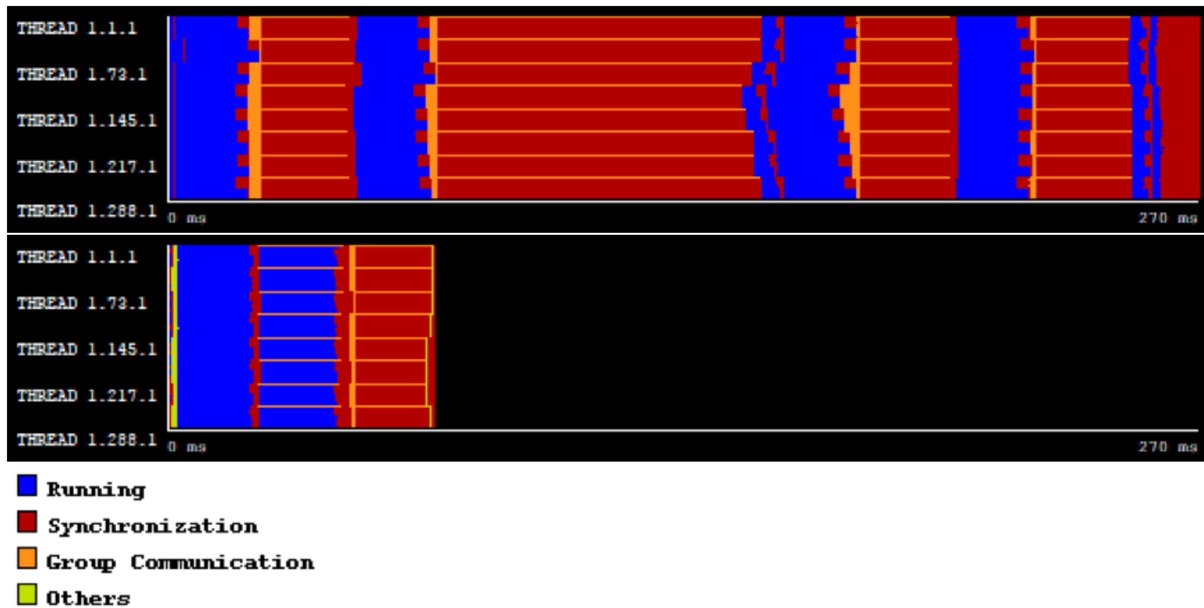

Figure 1: A single call to the multiplication subroutine using original code (top) and modified code (bottom) on 8 compute nodes, matrices are of size 3200x3200.

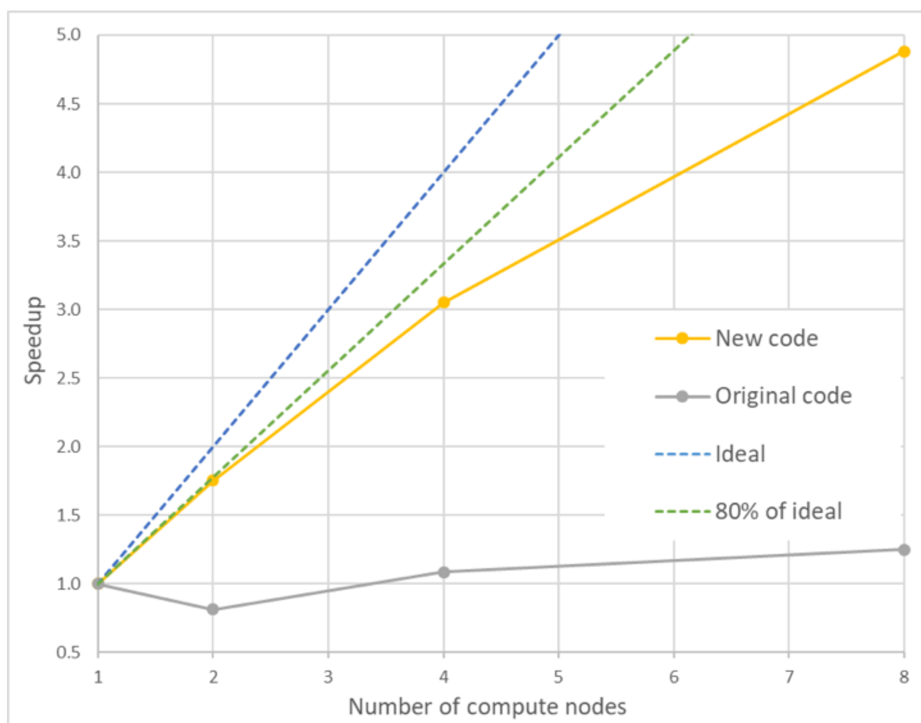

Figure 2: Speed up for the new and original code, all data is relative to the original code on 1 compute node, matrices are 3200x3200.

3.5 Conclusion

An optimised version of the shared array method was implemented within the ADF source code which showed a large improvement over the original code. Computational scaling was improved by using a better partitioning strategy for the dgemm calls, and parallel efficiency was improved by overlapping communication with computation and removing unnecessary calls to MPI_Allgatherv. Further improvements were obtained by reusing temporary storage.

This work nicely shows the importance of understanding performance of BLAS and PBLAS when designing parallel algorithms. In this particular case the original shared array method computed blocks that were increasingly rectangular as the number of processes was increased, and this was found to be highly inefficient. It was also interesting that the MPI version of pzgemm outperformed hybrid (MPI + OpenMP) versions, and that improving the load balance of pzgemm introduced worse serialisation and transfer efficiency, with no overall improvement.

This work also nicely shows the importance of overlapping computation and communication in order to get good parallel performance.

The slow performance of MPI and BLAS for freshly created POSIX shared memory arrays was unknown to SCM, and while this may be simply due to overheads of creating and initialising storage, it could also be caused by NUMA (non-uniform memory access) issues, as the hardware used had two NUMA nodes per compute node.

4 Report on Activity uhisolver

4.1 Description of the Application

uhiSolver (Urban Heat Island Solver) is a program to forecast local conditions (like thermal comfort) during the hottest days of summer in densely built urban areas including the cooling effects of plants and water surfaces due to evaporation. Initially, the program has been known to POP as chtMollierSolver. The software used for simulations was OpenFOAM-v1612+ with OpenMPI-1.6.5 both compiled using gcc-4.8.4.

uhiSolver calculates and models air-flow with day/night cycles, sun movement across the sky including direct and diffuse radiation as well as reflections, different surfaces albedos, buoyancy effects in air flow and evaporative cooling.

4.2 Previous Assessments and Recommendations

A POP Audit has already taken place, POP_AR_66. It found that overall the application focus of analysis (FOA) scaled well, with super-linear scaling due to improvement of the IPC (instructions per cycle).

Load Balance was the most significant bottleneck and was found to be due to reduced IPC on some ranks caused by increased cache misses. On 128 cores the Load Balance was only 67.4%. Both the Serialization and Transfer Efficiencies were over 80% with much less potential for improvement.

The recommendation that will be addressed and tested in this report is to improve the load balance by improving the locality of data and reducing the cache misses.

4.3 PoC Activities

The CFD simulation mesh used in the simulation gets decomposed into cells (domains). The new implementation renumbers the cell list during the initialisation phase. The aim is to improve the locality of the data and hence cache usage.

4.4 Results and Conclusions

- Load balance has successfully been improved by renumbering the mesh. On 128 cores the Load Balance metric has improved from 67.7% to 89.7%.
- Overall the performance of the application has improved by about 25% due to:
  - Improvement in the load balance leading to a performance gain of around 22%.
  - A small reduction in the number of instructions executed
  - This is very close to the maximal improvement that could be gained and constitutes a successful PoC.
- Serialization and dependencies in the communication are now the largest bottleneck in the application.
- Small sections of computation in between communication occasionally take longer than usual due to low CPU frequency.
- The same section also has an increased cache miss rate.

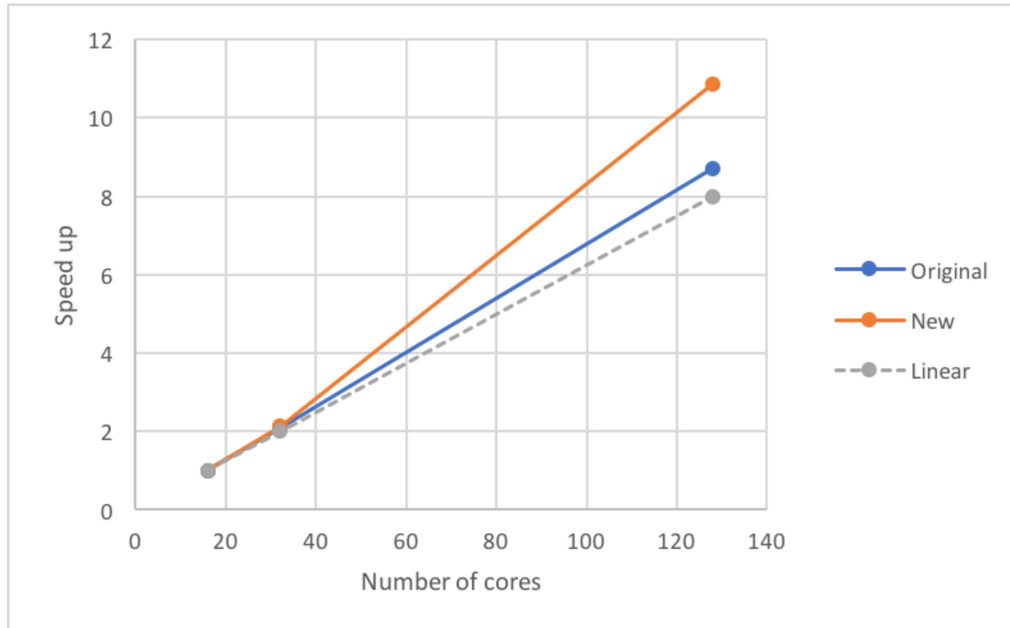

Figure 3: Some results.

- Further investigation of whether this is due to I/O would need re-installation of Extrae and new traces but there is more limited potential for improvement.

This PoC activity also shows nicely that cache-aware is necessary for good single-core performance.

5 Report on Activity zCFD

Keywords: OpenMP; Python

5.1 Description of the Application

zCFD by Zenotech (https://zcfd.zenotech.com/) is a density based finite volume and Discontinuous Galerkin (DG) computational fluid dynamics (CFD) solver for steady-state or time-dependent flow simulation. It decomposes domains using unstructured meshes. zCFD comprises a Python package (`zCFD-driver`) that calls computational kernels written in C++. This PoC investigated the performance of the C++ part of the DG solver only.

5.2 Previous Assessments and Recommendations

In the POP Performance Audit of zCFD we observed that the Parallel Efficiency was the main source of inefficiency. The Computational Efficiency was generally acceptable except on 12 threads, but the drop in performance at that thread count was mainly due to the CPU reducing its operating frequency to cope with the demand on resources. We recommended that further work should be conducted to:

1. Locate the portions of serial code in the Region of Interest (RoI) and identify which are potential candidates for parallelisation. This was the area in which the Audit suggested there was the largest single opportunity for improvement.
2. Investigate from where `operator()$omp$parallel:6@unknown:627:635` is called and whether there is anything that can be done to improve its load balance. This may involve changes in the distribution of work at a higher level or altering the scheduling strategy.
3. Ascertain whether the code is fully exploiting vectorisation, and if not determine what possible improvements could be implemented. Intel's Vectorization Advisor would provide useful guidance for this, as would the Intel compiler optimisation reports.

This PoC looked at points 1 and 2 only, although the compiler optimisation reports mentioned in point 3 were also used to guide the optimisations.

5.3 PoC Activities

Zenotech made a number of changes to the code as a result of the Audit:

1. Parallelising serial portions of code. The Audit identified that an appreciable fraction of the runtime was spent in serial code (c. 33% on 12 threads). Zenotech discovered that although the code contained the correct OpenMP pragmas, the Intel compiler found a particular region too complex to analyse and so did not apply any optimisations or OpenMP pragmas. This was solved by removing an `inline` keyword, which resulted in smaller blocks of code that the compiler was able to optimise. This also ensured that the OpenMP pragmas were enabled.
2. Improving load balance by code changes and different OpenMP loop scheduling policies. The Audit found that the main load imbalance occurred in the region of code responsible for computing the far-field boundary conditions and was due to a call to `pow()` hitting a slow code-path. Because both the base and the exponent were close to 1, `pow()` was

computing the result to high accuracy, which took a lot of time, but this accuracy was not needed by the computation. This was resolved by scaling the base, raising it to the power, and then undoing the scaling. Zenotech also experimented with different OpenMP loop scheduling policies and found that switching to `dynamic` improved load balance at the cost of some runtime overhead. They also implemented a scheme for dynamically setting the chunk size to avoid starving threads when the number of loop iterations was low.

3. Removing OpenMP regions that were being created on multiple threads. zCFD uses the `ThreadWorker` library. By default there are two worker threads and most of the OpenMP calls are performed by the first of these. In some cases, however, zCFD was also creating OpenMP parallel regions on the second thread. The Intel OpenMP runtime splits the thread pool equally between the two regions, but the work being done on the second thread was often minimal (e.g. unpacking packed MPI types in the case of the hybrid MPI-OpenMP version of the code). Furthermore, threads in the second region were kept alive by the runtime between OpenMP regions to avoid the overhead of repeatedly creating and destroying them. This meant that only half the CPU cores were actually doing useful computational work. Zenotech altered the code to remove the OpenMP region from the second worker thread, and this made all threads available to perform useful computation.

4. Memory management modifications. The original version of zCFD used Intel MKL's batch BLAS interface in an effort to minimise the overhead of making many BLAS calls with small matrices. This meant, however, that the code was spending a lot of time allocating and deallocating the additional arrays needed to perform the batch BLAS calls. Zenotech therefore re-engineered the code to call optimised small matrix kernels that skips the error checks the MKL usually performs. This was achieved using the `MKL_DIRECT_CALL` preprocessor macro. In addition it was hoped that this would improve cache utilisation as the BLAS calls are often made soon after the matrix has been populated and so should still be resident in cache.

5. Changing execution environment settings to boost CPU performance. The Audit found that the CPU frequency dropped to 90% of its base value when 12 threads were used, which had an noticeable effect on the performance of the code. Zenotech determined that the CPU frequency governor was set to `ondemand` by default on the machine used for the Audit, and that this was responsible for the drop in operating frequency. Adding `--cpu-freq=performance` to the `slurm` commands resolved the issue by instructing the CPU to run at its base frequency even when fully populated with threads.

5.4 Results

We recalculated the POP metrics for the modified code; these are presented in Table 1 alongside the metrics from the Audit. We note:

- Global, Parallel and Computational Efficiencies are all higher for the modified code than in the Audit on all thread counts.
- IPC Efficiency is worse for the modified code than in the Audit. The absolute IPC values (not shown) are lower as well: 1.62 on 12 threads versus 1.82 in the Audit.
- Instructions Efficiency on 12 threads is worse for the modified code than in the Audit, although the absolute number of instructions on 1, 2 and 6 threads is actually lower. The extra instructions executed on 12 threads are almost all located in the scheduling functions in the `vmlinux` kernel module.

	Original Code				Modified Code			
	# Threads				# Threads			
	1	2	6	12	1	2	6	12
Global Efficiency	97%	71%	52%	33%	100%	89%	73%	56%
Parallel Efficiency	97%	80%	64%	50%	100%	98%	89%	76%
Computational Efficiency	100%	89%	82%	66%	100%	91%	82%	74%
IPC Efficiency	100%	94%	92%	91%	100%	89%	88%	83%
Instructions Efficiency	100%	100%	100%	100%	100%	99%	99%	85%
CPU Frequency Efficiency	100%	94%	89%	72%	100%	104%	94%	105%
% Parallel Code	–	89%	75%	67%	–	99%	97%	86%
Load Balance Efficiency	100%	88%	85%	85%	100%	98%	89%	76%
OpenMP Overhead Efficiency	100%	100%	99%	98%	100%	100%	99%	99%

Table 1: POP efficiencies for the original and modified codes.

- The CPU Frequency Efficiency for the modified code is higher than in the Audit, almost certainly because of the change to the CPU frequency governor environment variable.
- The percentage of runtime spent in parallel code is higher than in the Audit. It is likely that there is little scope for additional parallelisation, with the remaining time in serial code comprising either the inevitable serial portions between parallel regions or, particularly on 12 threads, the scheduling functions in the `vmlinux` kernel module.
- Load Balance efficiency is better on 2 and 6 threads but is worse on 12. There was no single OpenMP region with extreme imbalance on 12 threads, however, and VTune identified the most imbalanced region as offering a potential reduction in runtime of 0.23 seconds (6% of total RoI runtime) if it were fixed.
- OpenMP overheads are unchanged, despite switching the loop scheduling policy to `dynamic`.

5.5 Conclusion

On 12 threads we found that the RoI in the modified code ran 1.65x faster than the RoI in the Audit code. The changes implemented in this PoC had even more of an impact for larger input cases, however, because these are more strongly affected by the load imbalance and high proportion of serial execution in the original code. Zenotech observed a 3x performance improvement on 12 threads for a test case 100x larger than the one used in this PoC, going from an average cycle time of 3 253ms to an average of 1 185ms. This corresponds to going from 10.356 GFlop/s to 30.631 GFlop/s for a single Broadwell socket.

We initially encountered problems profiling zCFD because the tool used was unable to recognise when the C++ kernels were called from Python. Given the increasing popularity of Python, developers of profiling tools should ensure that their applications work fully with the language.

6 Report on Activity DNS JET

Keywords: MPI, OpenMP, distributed FFT, domain decomposition, FFTW, 2DECOMP&FF

6.1 Description of the Application

DNS JET is a finite-difference code used for simulation of jets in astrophysics and engineering. The code is parallelised with MPI and OpenMP. The DNS JET application uses a 1D decomposition for input and output domains. All arrays were split into slices along the Z-axis or the Y-axis manually. Each MPI process has only one slice of each array. The change in the arrangement of arrays by processes occurs with the help of MPI_Alltoall.

6.2 Previous Assessments and Recommendations

The DNS JET application has already been analysed in detail in the POP Audit Report (POP_AR_88). The analysis revealed the following issues:

- The strong scaling experiment for MPI shows bad speedup inside a single node as well as across several nodes.
- The variation of the number of OpenMP threads has almost no effect on the performance of the application.
- The application with 180 MPI processes has a low communication efficiency. The reason is that the application spends more than 50 % on MPI collective communications. The most important MPI function is MPI_Alltoall.
- There are two sequential routines, namely `m_fft.fftib_` and `m_fft.fftdb_`, which take more than 30 % of the runtime in the region of interest and, therefore, they should be parallelised.

6.3 PoC activities

An appreciable fraction of the application time is spent on FFT operations. In the original, these operation were using serial FFTW routines. A first effort was done to invoke hybrid OpenMP/MPI versions of FFTW. This led to a performance increase of roughly 7% for the given input data set. The domain decomposition was 1D which seemed inefficient for larger problems sizes. Therefore, a second phase of the PoC concentrated on choosing domain decomposition, i.e. 1D or 2D, in a semi-automatic way in order to achieve good results across a range of different problems sizes.

In order to change the 1D domain decomposition to 2D decomposition we used the 2DECOMP&FFT library. It is designed for applications using three-dimensional structured mesh and spatially implicit numerical algorithms. At the foundation it implements a general-purpose 2D pencil decomposition for data distribution on distributed-memory platforms. On top it provides a highly scalable and efficient interface to perform three-dimensional distributed FFTs. The library is optimised for supercomputers and scales well to hundreds of thousands of cores.

Using 2DECOMP&FFT requires significant, but repetitive, refactoring of the code and data structures. Therefore, is was decided to demonstrate the steps on a mockup of the code, rather than working on the original code. This mockup is referred to as prototype in the remainder of this report.

Description	Original code		Prototype	
	Name	Time (sec.)	Name	Time (sec.)
Creating plan	fftw_setup_many()	0.14	-	-
Forward FFT (r2c)	fftdb()	0.13	decomp_2d_fft_3d()	0.44
Backword FFT (c2r)	fftid()	0.13	decomp_2d_fft_3d()	0.48
Rotation (Z $\rightarrow$ Y)	ParS2P()	0.37	transpose_z_to_y()*	0.12
Rotation (Y $\rightarrow$ Z)	ParP2S()	0.39	transpose_y_to_z()*	0.16
Total time (sec.)	-	1.02	-	0.92
Speedup	-	1	-	1.11

Table 2: Executing time of separate subroutines for the original code and the prototype of DNS JET. Note, that routines marked with a superscript * are not implemented in the prototype.

The code optimisations were evaluated on the customers target system MARCONI A1 at CINECA using total execution time as performance metric.

6.4 Results

A summary of the code modification points including the subroutines for the original code and the prototype of DNS JET are shown in Table 2. Both applications used a single precision for all domains.

To evaluate the made improvements we compare average time for 1 MPI rank for one internal iteration in the `rkutta` subroutine and for 1 input domain (`cur`). For the original code, one internal iteration in the `rkutta` subroutine includes the following functions: `fftdb()`, `fftid()`, `ParS2P()` and `ParP2S()`. For the prototype, one internal iteration contains only 2 calls of `decomp_2d_fft_3d()` for forward and backward FFTs. The time for creating the plan is not taken into account in both cases.

Thus, this time for one internal iteration in the `rkutta` subroutine is 1.02 seconds for the original code and 0.92 seconds for the prototype. It means, that the speedup equals to 1.11. Average time for 1 MPI rank and a full loop in the `shear` subroutine for 50 iterations is 153.33 seconds for the original code and 138.11 seconds for the prototype. The difference is 15.22 seconds or 9.93 %.

The auto-tuning algorithm for selection of the processor grid used for this test a `1*32` process grid, which is effectively the same as the 1D slab decomposition. So the only performance gain for this test is from the carefully written code that pack and unpack the send and receive buffers for the MPI_Alltoallv communication. The benefit of the library will normally becomes more obvious for larger problem sizes.

6.5 Conclusion

This PoC focused on the domain decomposition. The DNS JET application uses a 1D decomposition for input and output domains. All domains were split into slices along the Z-axis or the Y-axis manually. Each MPI process had only one slice of each array. The change in the arrangement of domains by processes occurs with the help of MPI_Alltoall, thus resulting in a large number of invocations. Besides, the 1D decomposition, while quite simple, has some limitations, especially for large-scale simulations.

In order to change the 1D domain decomposition to 2D decomposition we used the 2DECOMP&FFT library. It is designed for applications using three-dimensional structured mesh and spatially implicit numerical algorithms. It implements a general-purpose 2D pencil decomposition for data distribution on distributed-memory platforms. It also provides a highly

scalable and efficient interface to perform three-dimensional distributed FFTs. The library is optimised for supercomputers and scales well up to hundreds of thousands of cores.

The prototype uses only the transform out of place and different type for physical and spectral spaces (`real` and `complex`). There is no need for the transformation by using the rotation. The use of the 2DECOMP&FFT library provides an easy way to build decomposition and the performance gain of up to 10 %.

7 Report on Activity Vampire

7.1 Description of the Application

Vampire is a general purpose code for calculating the equilibrium and dynamic magnetic properties of magnetic materials. It is designed to cope with a wide range of magnetic phenomena, including spin glass, ferro, ferri and antiferromagnets, and magnetic anisotropies. It can simulate a variety of structural media, including bulk systems, thin films, nanoparticles, and various crystal structures.

Vampire is written in C++ and parallelised using MPI.

7.2 Previous Assessments and Recommendations

A POP audit has already taken place, POP_AR_9. It found that the application scaled well on a small number of cores and, in the region of interest (ROI), the distribution of computation across MPI processors was very well balanced, with a measured load balance on four processors of 97.91%.

The communication efficiency was slightly lower, at 83.14%, suggesting there could be some room for improvement in the efficiency of the communication between processes. Specifically, the time spent receiving messages was found to be quite high and imbalanced across the processors. These issues have been investigated in a POP performance plan, POP_PP_13, which further investigated the performance of the code on multiple nodes.

The POP audit also investigated the serial performance of the code and found that it could benefit significantly from increased use of vectorisation. It is this issue that was the focus of the Proof of Concept report.

7.3 PoC activities

We implemented the following modifications to speed up the Vampire code.

- The random number generator calls to the Intel MKL Vector Statistics Library for vectorised random number generation.
- There are several loops (most notably those in the runtime but are not vectorised because of data dependencies assumed by the compiler to be present. We used Intels Vector Advisor tool to check for data dependencies. Where no dependencies were found, we forced the compiler to vectorise loops using the `#pragma omp simd` directive.
- We investigated whether enforcing data alignment, and refactoring some arrays and loops for improved cache efficiency could provide performance improvements.

7.4 Results and Conclusion

The code changes were tested on an Intel Xeon 6148 processor containing 20 cores, and providing up to 40 hardware threads. Results for one test case are shown in the figure above. Depending on the number of processes, switching to vectorised random number generators resulted in up to a 27% performance improvement. When the `#pragma omp simd` directive was used to enforce loop vectorisation, up to an additional 15% improvement was obtained. Data alignment and array refactoring resulted in minimal improvements to the runtime.

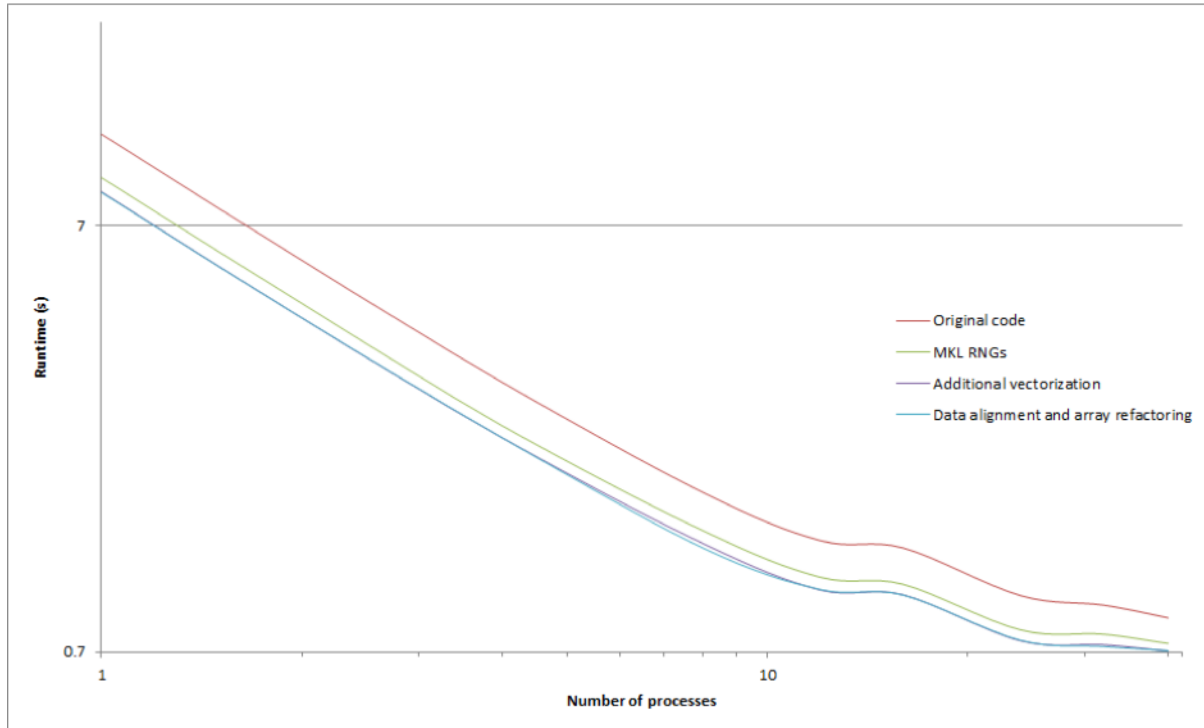

Figure 4: Runtimes comparing the original Vampire code with various code modifications.

Overall, up to a 46% performance improvement in runtime was achieved over the original code, though this number varied according to the test case and the number of processes. A drop off in scaling performance on higher core counts was found to be caused by memory bandwidth contention.

8 Report on Activity Reveal

8.1 Description of the Application

This report will outline the POP analyses of the Reveal seismic analysis code. The Reveal code was analysed in a POP audit (POP_AR_77 under the application name OpenCPS), POP performance plan (POP_PP_14), and finally a POP proof of concept (POP_PoCR_17). Reveal is a seismic processing code that does time and depth analysis for land and maritime applications. Reveal land processing tools cover all aspects of land processing from refraction statistics to final time and depth imaging.

8.2 Previous Assessment and Recommendations

The POP performance plan identified that Reveal had good load balance and good Amdahl's efficiency (percentage of parallel execution). It also identified poor parallel efficiency and this is due to the OpenMP overheads caused by a critical section in the most computationally intensive OpenMP parallel region. The pseudocode of this OpenMP parallel region is shown in Listing 1 and the critical section is shown in line 10.

```
#pragma omp parallel
{
#pragma omp for collapse(3)
  for ( int i = 0; i < Ni; i++ ) {
    for ( int j = 0; j < Nj; j++ ) {
      for ( int k = 0; k < Nk; k++ ) {
        float PTime[N], POut[N]; // thread local variable

        // blocks of code doing computation
#pragma omp critical
{
        get( POut[:] );
        PTime[:] = PTime[:] * weight;
        POut[:] += PTime[:];
        put( POut[:] );
} // end omp critical

      } // end k
    } // end j
  } // end i
} // end omp parallel
```

Listing 1: Reveal pseudocode for the focus of analysis

The critical was probably placed to prevent race conditions during read and write I/O operations. The recommendation in the performance plan was to restructure the code so that the OpenMP critical is not required.

8.3 Implementation and Results

To restructure the code so that the critical section is not required, the following steps were taken:

- A large multi-dimensional array was used to store reduction operations on line 14 in Listing 1. The allocation of the array occurred before line 1 and deallocation occurred after 21;
- The read and write I/O operations (lines 12 and 15) were taken outside the OpenMP parallel region, namely after line 21.

The speed-up graph is shown in Figure 5, where the modified PoC code uses both static and dynamic OpenMP schedules, and the original code used the static schedule. Performance gains are only achieved for 18 and 24 threads. This is due to reduced Amdahl's law efficiency caused by the allocation/deallocation of the multi-dimensional array and the read/write file I/O operations.

The impact of the memory allocation/deallocation and file I/O was estimated by removing them completely. This will result in an incorrect solution, but since PoC did not have access to the full code to implement them completely it shows the benefit of further code changes. The resulting speed-up graph is shown in Figure 6 where the linear and 80% of linear graphs are scaled by the variable CPU frequency[1]. As can be seen from Figure 6, the yellow line shows the scaling (memory allocation/deallocation and file I/O removed) will yield better than 80% of linear scalability.

8.4 Conclusion

The original code's performance was hindered by the file I/O operations enclosed by an OpenMP critical region. The recommendation was to restructure the code so it does not require the critical region, but additional performance gains can be made by removing the file I/O and applying the OpenMP dynamic schedule. Final recommendations were made to move the memory allocation/deallocation outside the function and pass the multi-dimensional array as a pointer argument and parallelise a sequential loop. The modified PoC code with the memory allocation/deallocation and file I/O removed shows a performance gain of 44% when compared with the PoC code with the memory and I/O operations. If the recommendations are implemented, the code is likely to see performance gains approaching closer to linear speed-up.

[1]The turbo boost feature adjusts the CPU frequency based on the load.

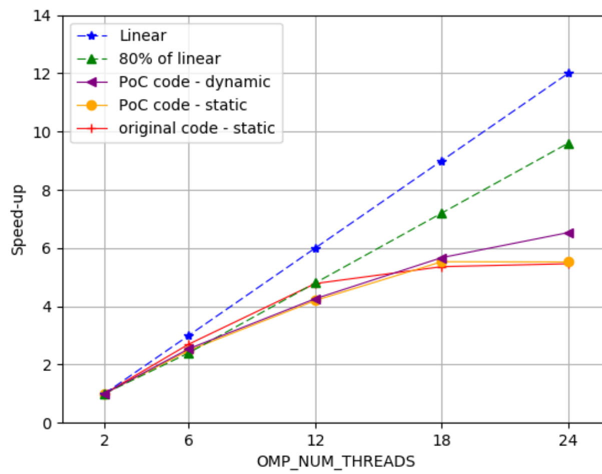

Figure 5: Speed-up of focus of analysis

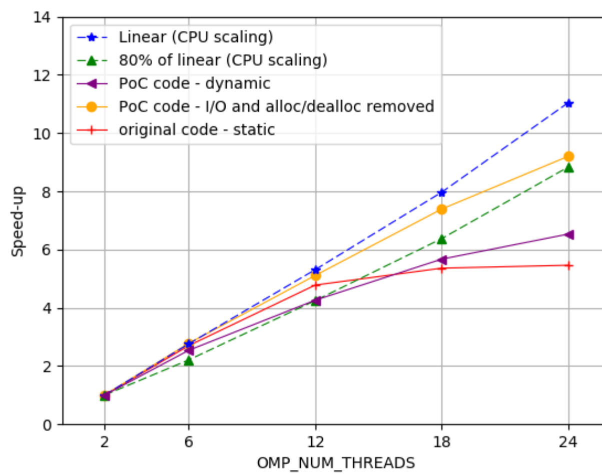

Figure 6: Potential speed-up of focus of analysis

9 Report on Activity SHEMAT Suite

9.1 Description of the Application

9.2 Previous Assessments and Recommendations

In POP_AR_21, a performance audit was performed for the SHEMAT suite. In the analysis, computational load imbalances were discovered in the `formapproxjacobian()` function. A later performance plan, POP_PP_03 identified process-to-core pinning to be responsible for the imbalance.

9.3 PoC activities

Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
▶ cap_pres	757,862,600,000	1,692,983,600,000	0.448
▶ mobility	300,066,800,000	575,159,200,000	0.522
▶ psi	247,546,200,000	436,994,800,000	0.566
▶ s_wr	237,248,000,000	859,273,800,000	0.276
▶ flow_term	198,499,400,000	568,537,200,000	0.349
▶ for_cpstr	127,545,000,000	342,089,000,000	0.373
▶ bc_lambda	121,462,000,000	397,768,800,000	0.305
▶ s_nr	114,659,600,000	413,773,800,000	0.277
▶ bc_pd	92,173,400,000	340,010,000,000	0.271
▶ upwind	83,831,000,000	198,994,400,000	0.421
▶ I_MPI_COLL_SHM_GENERIC_RELEASE_BCA	66,059,400,000	26,085,400,000	2.532
▶ hypre_BoomerAMGBuildCoarseOperatorKT	48,862,000,000	118,943,000,000	0.411
▶ f_12	46,961,200,000	148,387,800,000	0.316
▶ arithmetic	44,041,800,000	149,289,800,000	0.295
▶ u_fct	41,494,200,000	134,807,200,000	0.308
▶ kx	35,277,000,000	126,031,400,000	0.280
▶ _intel_fast_memcmp	34,940,400,000	61,545,000,000	0.568
▶ MatSetValues_MPIAIJ	34,656,600,000	82,084,200,000	0.422
▶ kz	30,593,200,000	115,827,800,000	0.264
▶ hypre_BoomerAMGRelax	29,902,400,000	77,171,600,000	0.387
▶ formapproxjacobian	28,322,800,000	47,645,400,000	0.594
▶ area_fv	21,164,000,000	87,291,600,000	0.242

Figure 7: A list of most time consuming functions in SHEMAT suite.

In this proof-of-concept, the SHEMAT suite was optimized to improve its overall execution time. The starting point of the work was exploring optimization opportunities in the most time-consuming functions of the code. The figure 7 below shows a list of functions called in the SHEMAT suite, sorted, in descending order, by their total execution time.

The figure shows that function cap_pres and mobility are the most time consuming. The proof-of-concept took a deeper look at these functions, and identified calls to several one-line functions. The calls to one-line functions not only added several jump statements in the assembly code, hence adding overhead to application execution, but also hindered the compiler to reorder assembly code to best utilize the available resources.

The figure 8 shows the code for the cap_pres function. The Fortran code in the figure shows calls to bc_lambda, bc_pd, S_wr and S_nr, which are all simple one-line functions. The assembly code on the right shows that the functions are not inlined by the compiler, but are called one-by-one. Additionally, the figure shows that the two calls to S_wr, with the same parameters, result in the function being called twice in the assembly code.

Line	Source	Count	Address	Line	Instruction
44	Smax=1.		0x46c0f6	52	mov %r15, %rsi
45	Smin=0.0000		0x46c0f9	52	mov %rbp, %rdx
46	dS = 0.01		0x46c0fc	52	mov %rbx, %rcx
47	N_l=101		0x46c0ff	52	xor %eax, %eax
48	N_S=102		0x46c101	52	callq 0x46b110 <bc_pd>
49			**0x46c106**		**Block 3:**
50	cap_pres = 0.0d0		0x46c106	52	movsdq %xmm0, 0x10(%rsp)
51	lambda = bc_lambda(i,j,k,ismpl)	16,	0x46c10c	54	mov %r12, %rdi
52	pd = bc_pd(i,j,k,ismpl)	13,	0x46c10f	54	mov %r15, %rsi
53			0x46c112	54	mov %rbp, %rdx
54	S_e = ((1.0d0 - satn(i,j,k,ismpl)) - S_wr(i,j,k,ismpl)) / &	145,	0x46c115	54	mov %rbx, %rcx
55	(1.0d0 -S_wr(i,j,k,ismpl) -S_nr(i,j,k,ismpl))	47,	0x46c118	54	xor %eax, %eax
56			0x46c11a	54	callq 0x479d50 <s_wr>
57	IF (bc_vg_mod=='bc') THEN		**0x46c11f**		**Block 4:**
58	if (lambda.gt.lamax) lambda=lamax		0x46c11f	54	movsdq %xmm0, 0x8(%rsp)
59	if (S_e.gt.Smax) S_e=Smax		0x46c125	55	mov %r12, %rdi
60	if (S_e.lt.Smin) S_e=Smin		0x46c128	55	mov %r15, %rsi
61			0x46c12b	55	mov %rbp, %rdx
62	dl = (S_e - Smin)/dS	33,	0x46c12e	55	mov %rbx, %rcx
63	dm = (lambda - lamin)/dlambda	12,	0x46c131	55	xor %eax, %eax
64			0x46c133	55	callq 0x479d50 <s_wr>
65	lp=dl		**0x46c138**		**Block 5:**
66	mp=dm	23,	0x46c138	55	movsdq %xmm0, (%rsp)

Figure 8: Fortran code and a snippet of the generated assembly code of the cap_pres function.

9.4 Results

Line	Source	Count	Address	Line	Instruction
43	dlambda=0.1		0x46c196	51	neg %rdx
44	Smax=1.		0x46c199	51	add %r13, %rdx
45	Smin=0.0000		0x46c19c	51	sub %r8, %rdx
46	dS = 0.01		0x46c19f	57	movsdq 0x21eaa1(%rip), %xmm2
47	N_l=101		0x46c1a7	42	movsdq 0x21ea91(%rip), %xmm0
48	N_S=102		0x46c1af	64	movsdq 0x21ea91(%rip), %xmm4
49			0x46c1b7	87	movsdq 0x21ea88(%rip), %xmm8
50	cap_pres = 0.0d0		0x46c1c0	51	leaq (%rdx,%rsi,8), %rdi
51	lambda = propunit(uindex(i,j,k),idx_bc_lambda,ismpl)	92,565,000,0...	0x46c1c4	52	mov %rax, %rdx
52	pd = propunit(uindex(i,j,k),idx_bc_pd,ismpl)	19,428,200,0...	0x46c1c7	60	minsdq (%r14,%rdi,1), %xmm12
53	S_wr = propunit(uindex(i,j,k),idx_s_wr,ismpl)	27,511,000,0...	0x46c1cd	52	shl $0x4, %rdx
54	S_nr = propunit(uindex(i,j,k),idx_s_nr,ismpl)	1,817,200,000	0x46c1d1	65	subsd %xmm0, %xmm12
55			0x46c1d6	53	add %rdx, %rax
56	S_e = ((1.0d0 - satn(i,j,k,ismpl)) - S_wr) / &	60,423,000,0...	0x46c1d9	52	sub %rdx, %r15
57	(1.0d0 -S_wr -S_nr)	9,246,600,000	0x46c1dc	53	neg %rax
58			0x46c1df	52	add %rcx, %r15
59	IF (bc_vg_mod=='bc') THEN		0x46c1e2	53	add %rcx, %rax
60	if (lambda.gt.lamax) lambda=lamax	352,000,000	0x46c1e5	52	neg %r15
61	if (S_e.gt.Smax) S_e=Smax		0x46c1e8	53	neg %rax
62	if (S_e.lt.Smin) S_e=Smin		0x46c1eb	52	add %r13, %r15
63			0x46c1ee	53	add %r13, %rax
64	dl = (S_e - Smin)/dS	8,800,000,000	0x46c1f1	52	sub %r8, %r15
65	dm = (lambda - lamin)/dlambda	7,189,600,000	0x46c1f4	53	sub %r8, %rax

Figure 9: The optimized version of cap_pres function. The assembly code shows compiler reordering of the assembly code for better resource utilisation.

The task of the proof-of-concept was then to manually inline the numerus one-line functions found in the SHEMAT suite. In total, seven different one-line functions were found. These functions accessed a multi-dimensional array, and some of them performed interpolation between the values. Therefore, it was possible to replace the function calls with the actual call to the multi-dimensional array, without reducing the generality of the software architecture.

The figure 9 shows the resulting code of the cap_pres function. In particular, it illustrates that the function calls have been replaced with access to the propunit array. Additionally, the assembly code in the figure shows that the compiler is now able to reorder the assembly code to efficiently utilize available resources. The mobility function was also updated accordingly.

The figure 10 below shows the most time-consuming functions after applying the optimisation. The figure shows that, the application not only saves in execution time by removal of the one-line functions, but due to better CPI and reduced redundancy in the code, the execution

Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
▶ cap_pres	614,176,200,000	1,636,652,600,000	0.375
▶ mobility	265,819,400,000	603,033,200,000	0.441
▶ psi	247,849,800,000	439,511,600,000	0.564
▶ flow_term	206,712,000,000	681,863,600,000	0.303
▶ for_cpstr	114,191,000,000	338,102,600,000	0.338
▶ upwind	61,171,000,000	145,358,400,000	0.421
▶ u_fct	46,552,000,000	154,440,000,000	0.301
▶ f_12	43,916,400,000	151,630,600,000	0.290
▶ arithmetic	43,780,000,000	141,717,400,000	0.309
▶ hypre_BoomerAMGBuildCoarseOperatorKT	43,012,200,000	104,559,400,000	0.411
▶ _intel_fast_memcmp	33,539,000,000	62,728,600,000	0.535
▶ MatSetValues_MPIAIJ	32,918,600,000	77,497,200,000	0.425
▶ hypre_BoomerAMGRelax	27,159,000,000	69,726,800,000	0.390
▶ formapproxjacobian	27,110,600,000	48,109,600,000	0.564
▶ area_fv	20,321,400,000	78,507,000,000	0.259
▶ ISLocalToGlobalMappingApply	16,350,400,000	31,253,200,000	0.523
▶ MatSetValuesStencil	15,175,600,000	35,488,200,000	0.428
▶ rhon	13,992,000,000	14,682,800,000	0.953
▶ I_MPI_COLL_SHM_GENERIC_RELEASE_B...	13,477,200,000	5,308,600,000	2.539
▶ copy_user_enhanced_fast_string	12,947,000,000	11,431,200,000	1.133
▶ ijk_m	12,817,200,000	11,462,000,000	1.118
▶ MatLUFactorNumeric_SeqAIJ_Inode	11,864,600,000	23,788,600,000	0.499

Figure 10: The most time-consuming functions of SHEMAT suite after optimisation.

time of cap_pres and mobility also reduces. As a result, the total execution time was reduced by 25%.

The next phase of the proof-of-concept tried to optimize the flow_term function, by inlining the psi function. However, the psi function, though performing a simple one line computation, itself calls other functions, hence hindering the compiler in reordering the assembly code even when inlined in flow_term. Therefore, the attempted optimization did not result in any concrete speedup.

9.5 Conclusion

The two most time-consuming functions of the SHEMAT suite called several one-line functions, which were not being inlined by the compiler. These one-line functions accessed a multi-dimensional array, and used the values for interpolation. Modifying the SHEMAT suite to accesses the array directly in the parent functions, and bypassing the one-line functions, resulted in the compiler generating a more optimized assembly code. As a result, the optimization lead to a 25% speedup in total execution time.

10 Report on Activity Relational Discovery

10.1 Description of the Application

The Relational Discovery application by Cybeletech processes relational databases (typically SQL) with a well-defined relational skeleton between tables. The idea is to discover complex statistical structures between columns of possibly different tables. The application can be viewed as an extension of Bayesian network discovery to relational databases and is used by the customer SME in the agricultural domain.

10.2 Previous Assessments and Recommendations

The MPI version of the application had been analysed in the POP_AR_72 assessment report. Load imbalance had been identified as the main issue in the application. A hybrid MPI+OpenMP approach and its combination with the Dynamic Load Balancing Library (DLB) by BSC was proposed as possible directions to explore.

10.3 PoC activities and Results

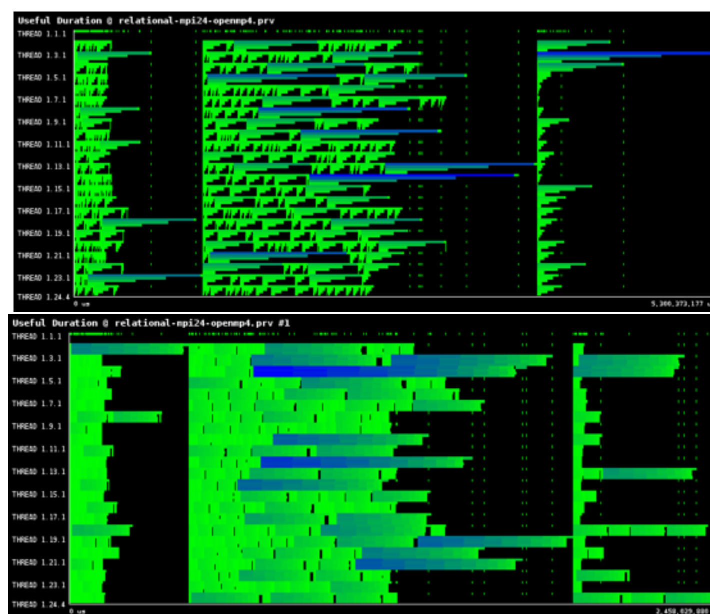

Figure 11: Original MPI+OpenMP (24x4) timeline (duration 5300s) on top; Dynamic scheduled MPI+OpenMP (24x4) timeline (duration 2458s) bottom.

In the POP_PoCR_16 we started from the analysis of an MPI+OpenMP version provided by the customer. The program had a single parallel loop and used a static scheduling at the OpenMP level which actually ended up in exposing the same load imbalance issues at the hybrid level. A simple transformation was to change the scheduling to dynamic, which certainly improved the OpenMP imbalance, but kept the imbalance at the MPI level. The behaviour is shown in Figure 11. The execution was performed in an ARM ThundeX1 based node with 96 cores.

To correct the MPI imbalance we used the DLB library. The library shifts cores between MPI processes in a node to avoid idling in one process while other processes have work. The

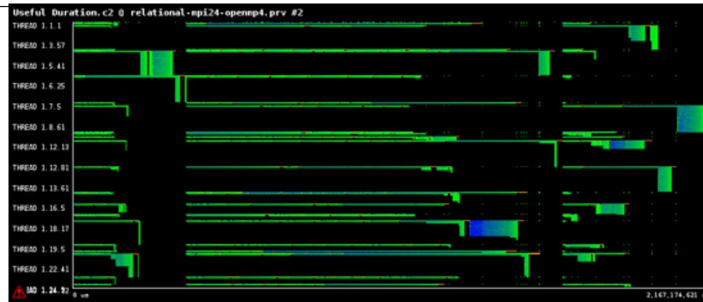

Figure 12: TImeline of a DLB based run (duration 1267s).

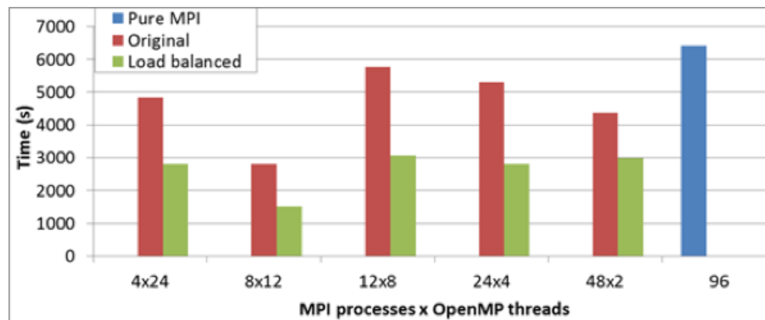

Figure 13: Performance comparison of the major versions of the code.

transfer between processes can be done automatically when the library detects that a process blocks in an MPI call but can also be steered by API calls where the programmer can specify points where a process can release its cores or when it claims them back.

A requirement of the approach is that the application must be malleable, that is, must have frequent point where the number of threads of a process can be changed. In the case of workshared OpenMP codes this is the point of the parallel pragma. The original program had three nested loops and was following a traditional approach where the parallels are put in the outermost loop. The granularity was very coarse and this resulted in very poor malleability, so the first refactoring was to move the parallel pragma to the second parallel loop. Maleability improved a lot and the granularity was still such that the OpenMP fork-join overhead was still irrelevant. We used the DLB API to hint the runtime when the program was entering the parallel region and leaving it to perform other non-parallelized computations. The behaviour is shown in Figure 12 where we appreciate how some processes are able to execute with many threads when other processes finish earlier at the end of each of the three computation phases.

Another observation can be made on the runs with the DLB approach. At the MPI level, the application devotes the first process to assign work to the other MPI processes and it only uses one thread for that. By using DLB the other processes were able to use during all the run the OpenMP worker threads that would otherwise stay idle.

The results are summarized in Figure 13 where we compare the execution times of the pure MPI code, the original MPI+OpenMP code provided by the customer and the optimized version with DLB for different configurations of processes and threads.

Further details on the behaviour of the different versions and an analysis of the results can be found in the POP_PoCR_16 report.

10.4 Conclusion

In total we developed 4 versions of the program and developed different variations of the runtime API. This served us as an important co-design experience as we found out that it is better not to follow very greedy policies where all resourced made available at one point in time should not be all reassigned to a single process. Instead, dumped mechanisms by which incremental shifts are supported proved being better. We also observed how the application itself can give hints of the maximum amount of cores it would need in a parallel region to avoid over allocating resources to processes that do not need them.

11 Conclusions and Lessons Learned

In this document we have reported the current state of Proof-of-Concept activities, which is the main service activity in Work Package 5. PoC is the highest level of service of the POP Project aiming at assisting customers to implement recommendations of previous POP Project performance assessments.

So far, the POP Project has concluded 19 PoC activities, with a further 3 in progress beyond the formal end of the project. These activities have in most cases led to significant improvement of the respective code's performance under the conditions define at the beginning of the activity. More importantly however, follow-up communication done under the Work Package 2 shows, that significant improvements are also realised under production conditions on arbitrary data sets.

Lessons Learned

One of the recurring issues in the PoC activities was high degrees of load imbalance, which immediately affects application performance. There is two main reasons for load imbalance: the first is algorithmic or true imbalance of problem size, while the second is related to data-locality.

Decomposing an algorithm or problem into well balanced subset is far from trivial. In particular, when load balance changes during the application. One case where runtime systems can help is transient load imbalance within a shared memory node, as was for instance observed in the code MNCP. In such cases it helps to reassigned computational resources temporarily from one process to another. In the PoC on MNCP, the library DLB (dynamic load balance) is used to temporarily shift cores from one MPI process to another, allowing hybrid MPI/OpenMP codes to adjust to transient imbalances. We would recommend to incorporate such techniques into all OpenMP and MPI runtime systems.

Load imbalances across nodes, are more difficult to handle. In particular, developers do not have adequate tools to transfer loads between MPI processes. Unfortunately, currently there is no approach that works well and more research is necessary.

The second origin of load imbalance is data-locality which translates into non-uniformity of data accesses from different compute resources and may thus lead to load imbalance if data is systematically mis-places. The topology and layout of memory hierarchies is a system property and thus beyond the control of the developer. Therefore it should be the responsibility of the runtime system do assign compute resources in such a way as to minimize non-local data access. For this to happen, however, the developer needs to give hints to the runtime which operation is going use which data. Currently, there is very little support from programming languages to express such relations.

Similarly, exploiting cache levels is of utmost importance to achieve high single-core performance. There is some vendor-specific compiler extensions that allow to instruct compiler to reorganize loop nests for better cache re-use. Clearly, this needs standardisation on the compiler or language level.

Regarding communication between MPI processes, we have seen that many applications suffer from high communication overheads, even if they are using non-blocking communication primitives or collective operations. Apparently, many MPI libraries do not drive progression of background transfers efficiently. MPI library developers seem to take the conservative approach and are reluctant to use compute resources to drive progression, as this impacts performance of compute loads. In this context, developers should be able to express at the language level, that they require background progression. Alternatively, interoperation between runtime systems,

e.g. OpenMP and MPI, would allow to share a core for such and similar background tasks thus minimising the impact on compute kernels. Applications that exhibit this problem in POP PoCs are BAND and Tangaroa. In both cases the gains expected from using non-blocking communications are less than anticipated.

The last major observation is that application developers reimplement standard parallel patterns on their own rather than using existing specialised libraries. One example is the usage of serial FFT libraries to do distributed FFT rather than resorting to a parallel, distributed FFT library. Also, developers often are not clear about the limitation os parallel libraries and use them in circumstances that result in poor performance. This situation can only be remedied by better training of parallel application developers and better documentation, in particular regarding limitations, of libraries.

Acronyms and Abbreviations

- BSC: Barcelona Supercomputing Center
- CA: Consortium Agreement
- CAdv: Customer Advocate
- DoA Description of Action (Annex 1 of the Grant Agreement)
- D: deliverable
- EC European Commission
- FFT: Fast-Fourier-Transform
- GA: General Assembly / Grant Agreement
- HLRS: High Performance Computing Centre (University of Stuttgart)
- HPC: High Performance Computing
- IPR Intellectual Property Right
- Juelich: Forschungszentrum Juelich GmbH
- KPI: Key Performance Indicator
- NUMA: non-uniform memory access
- M: Month
- MPI: Message-Passing Interface, a shared-memory programming model
- MS: Milestones
- OpenMP: a shared-memory programming model
- PEB: Project Executive Board
- PM: Person month / Project manager
- PoC: Proof-of-Concept
- POP: Performance Optimization and Productivity
- R: Risk
- RV: Review
- RWTH Aachen: Rheinisch-Westfaelische Technische Hochschule Aachen
- USTUTT (HLRS): University of Stuttgart
- WP: Work Package
- WPL: Work Package Leader